1 Using TEX

Turning input into ink

Programs and files you need

In order to produce a TEX document, you'll need to run the TEX program and several related programs as well. You'll also need supporting files for TEX and possibly for these other programs. In this book we can tell you about TEX, but we can't tell you about the other programs and the supporting files except in very general terms because they depend on your local TEX environment. The people who provide you with TEX should be able to supply you with what we call *local information*. The local information tells you how to start up TEX, how to use the related programs, and how to gain access to the supporting files.

Input to TEX consists of a file of ordinary text that you can prepare with a text editor. A TEX input file, unlike an input file for a typical word processor, doesn't ordinarily contain any invisible control characters. Everything that TEX sees is visible to you too if you look at a listing of the file.

Your input file may turn out to be little more than a skeleton that calls for other input files. TEX users often organize large documents such as books this way. You can use the `\input` command (p. '`\input`') to embed one input file within another. In particular, you can use `\input` to incorporate files containing *macro definitions*—auxiliary definitions that enhance TEX's capabilities. If any macro files are available at your TEX installation, the local information about TEX should tell you how to get

at the macro files and what they can do for you. The standard form of TEX, the one described in this book, incorporates a collection of macros and other definitions known as plain TEX (p. '`\plainTeX`').

When TEX processes your document, it produces a file called the `.dvi` file. The abbreviation "`dvi`" stands for "device independent". The abbreviation was chosen because the information in the `.dvi` file is independent of the device that you use to print or display your document.

To print your document or view it with a *previewer*, you need to process the `.dvi` file with a *device driver* program. (A previewer is a program that enables you to see on a screen some approximation of what the typeset output will look like.) Different output devices usually require different device drivers. After running the device driver, you may also need to transfer the output of the device driver to the printer or other output device. The local information about TEX should tell you how to get the correct device driver and use it.

Since TEX has no built-in knowledge of particular fonts, it uses *font files* to obtain information about the fonts used in your document. The font files should also be part of your local TEX environment. Each font normally requires two files: one containing the dimensions of the characters in the font (the *metrics file*) and one containing the shapes of the characters (the *shape file*). Magnified versions of a font share the metrics file but have different shape files. Metrics files are sometimes referred to as `.tfm` files, and the different varieties of shape files are sometimes referred to as `.pk` files, `.pxl` files, and `.gf` files. These names correspond to the names of the files that TEX and its companion programs use. For example, `cmr10.tfm` is the metrics file for the `cmr10` font (10-point Computer Modern Roman).

TEX itself uses only the metrics file, since it doesn't care what the characters look like but only how much space they occupy. The device driver ordinarily uses the shape file, since it's responsible for creating the printed image of each typeset character. Some device drivers need to use the metrics file as well. Some device drivers can utilize fonts that are resident in a printer and don't need shape files for those fonts.

Running TEX

You can run TEX on an input file `screed.tex` by typing something like '`run tex`' or just '`tex`' (check your local information). TEX will respond with something like:

```
This is TeX, Version 3.0 (preloaded format=plain 90.4.23)
**
```

The "preloaded format" here refers to a predigested form of the plain TEX macros that come with TEX. You can now type '`screed`' to get TEX to process your file. When it's done, you'll see something like:

```
(screed.tex [1] [2] [3] )
Output written on screed.dvi (3 pages, 400 bytes).
Transcript written on screed.log.
```

displayed on your terminal, or printed in the record of your run if you're not working at a terminal. Most of this output is self-explanatory. The numbers in brackets are page numbers that TEX displays when it ships out each page of your document to the `.dvi` file. TEX will usually assume an extension '`.tex`' to an input file name if the input file name you gave doesn't have an extension. For some forms of TEX you may be able to invoke TEX directly for an input file by typing:

```
tex screed
```

or something like this.

Instead of providing your TEX input from a file, you can type it directly at your terminal. To do so, type '`\relax`' instead of '`screed`' at the '`**`' prompt. TEX will now prompt you with a '`*`' for each line of input and interpret each line of input as it sees it. To terminate the input, type a command such as '`\bye`' that tells TEX you're done. Direct input is sometimes a handy way of experimenting with TEX.

When your input file contains other embedded input files, the displayed information indicates when TEX begins and ends processing each embedded file. TEX displays a left parenthesis and the file name when it starts working on a file and displays the corresponding right parenthesis when it's done with the file. If you get any error messages in the displayed output, you can match them with a file by looking for the most recent unclosed left parenthesis.

For a more complete explanation of how to run TEX, see Chapter 6 of *The TEXbook* and your local information.

Preparing an input file

In this section we explain some of the conventions that you must follow in preparing input for TEX. Some of the information given here also appears in the examples in Section 'examples' of this book.

Commands and control sequences

Input to TEX consists of a sequence of commands that tell TEX how to typeset your document. Most characters act as commands of a particularly simple kind: "typeset me". The letter '`a`', for instance, is a command to typeset an 'a'. But there's another kind of command—a *control sequence*—that gives TEX a more elaborate instruction. A control sequence ordinarily starts with a backslash (\), though you can change that convention if you need to. For instance, the input:

```
She plunged a dagger (\dag) into the villain's heart.
```

contains the control sequence `\dag`; it produces the typeset output:

She plunged a dagger (†) into the villain's heart.

Everything in this example except for the `\dag` and the spaces acts like a "typeset me" command. We'll explain more about spaces on page '`spaces`'.

There are two kinds of control sequences: *control words* and *control symbols*:

- A control word consists of a backslash followed by one or more letters, e.g., '`\dag`'. The first character that isn't a letter marks the end of the control word.
- A control symbol consists of a backslash followed by a single character that isn't a letter, e.g., '`\$`'. The character can be a space or even the end of a line (which is a perfectly legitimate character).

A control word (but not a control symbol) absorbs any spaces or ends of line that follow it. If you don't want to lose a space after a control word, follow the control sequence with a control space (\␣) or with '`{}`'. Thus either:

```
The wonders of \TeX\␣shall never cease!
```

or:

```
The wonders of \TeX{} shall never cease!
```

produces:

The wonders of TEX shall never cease!

rather than:

The wonders of TEXshall never cease!

which is what you'd get if you left out the '`\␣`' or the '`{}`'.

Don't run a control word together with the text that follows it—TEX won't know where the control word ends. For instance, the `\c` control sequence places a cedilla accent on the character that follows it. The French word *garçon* must be typed as '`gar\c␣con`', not '`gar\ccon`'; if you write the latter, TEX will complain about an undefined control sequence `\ccon`.

A control symbol, on the other hand, doesn't absorb anything that follows it. Thus you must type '$13.56' as '`\$13.56`', not '`\$␣13.56`'; the latter form would produce '$ 13.56'. However, those accenting commands that are named by control symbols are defined in such a way that they produce the effect of absorbing a following space. Thus, for example, you can type the French word *déshabiller* either as '`d\'eshabiller`' or as '`d\'␣eshabiller`'.

Every control sequence is also a command, but not the other way around. For instance, the letter '`N`' is a command, but it isn't a control sequence. In this book we ordinarily use "command" rather than "control sequence" when either term would do. We use "control sequence" when we want to emphasize aspects of TEX syntax that don't apply to commands in general.

Arguments

Some commands need to be followed by one or more *arguments* that help to determine what the command does. For instance, the `\vskip` command, which tells TEX to skip down (or up) the page, expects an argument specifying how much space to skip. To skip down two inches, you would type '`\vskip 2in`', where `2in` is the argument of `\vskip`.

Different commands expect different kinds of arguments. Many commands expect dimensions, such as the `2in` in the example above. Some commands, particularly those defined by macros, expect arguments that are either a single character or some text enclosed in braces. Yet others require that their arguments be enclosed in braces, i.e., they don't accept single-character arguments. The description of each command in this book tells you what kinds of arguments, if any, the command expects. In some cases, required braces define a group (see p. '`bracegroup`').

Parameters

Some commands are parameters (p. ‘parameter’). You can use a parameter in either of two ways:

1) You can use the value of a parameter as an argument to another command. For example, the command `\vskip\parskip` causes a vertical skip by the value of the `\parskip` (paragraph skip) glue parameter.
2) You can change the value of the parameter by assigning something to it. For example, the assignment `\hbadness=200` causes the value of the `\hbadness` number parameter to be 200.

We also use the term “parameter” to refer to entities such as `\pageno` that are actually registers but behave just like parameters.

Some commands are names of tables. These commands are used like parameters, except that they require an additional argument that specifies a particular entry in the table. For example, `\catcode` names a table of category codes (p. ‘category+code’). Thus the command `\catcode‘~=13` sets the category code of the ‘~’ character to 13.

Spaces

You can freely use extra spaces in your input. Under nearly all circumstances TEX treats several spaces in a row as being equivalent to a single space. For instance, it doesn't matter whether you put one space or two spaces after a period in your input. Whichever you do, TEX performs its end-of-sentence maneuvers and leaves the appropriate (in most cases) amount of space after the period. TEX also treats the end of an input line as equivalent to a space. Thus you can end your input lines wherever it's convenient—TEX makes input lines into paragraphs in the same way no matter where the line breaks are in your input.

A blank line in your input marks the end of a paragraph. Several blank lines are equivalent to a single one.

TEX ignores input spaces within math formulas (see below). Thus you can include or omit spaces anywhere within a math formula—TEX doesn't care. Even within a math formula, however, you must not run a control word together with a following letter.

If you are defining your own macros, you need to be particularly careful about where you put ends of line in their definitions. It's all too easy to define a macro that produces an unwanted space in addition to whatever else it's supposed to do. We discuss this problem elsewhere since it's somewhat technical; see page ‘unwantedspace’.

A space or its equivalent between two words in your input doesn't simply turn into a space character in your output. A few of these input spaces turn into ends of lines in the output, since input lines generally don't correspond to output lines. The others turn into spaces of variable width called "glue" (p. 'glue'), which has a natural size (the size it "wants to be") but can stretch or shrink. When TEX is typesetting a paragraph that is supposed to have an even right margin (the usual case), it adjusts the widths of the glue in each line to get the lines to end at the margin. (The last line of a paragraph is an exception, since it isn't ordinarily required to end at the right margin.)

You can prevent an input space from turning into an end of line by using a tie (`~`). For example, you wouldn't want TEX to put a line break between the 'Fig.' and '8' of 'Fig. 8'. By typing '`Fig.~8`' you can prevent such a line break.

Comments

You can include comments in your TEX input. When TEX sees a comment it just passes over it, so what's in a comment doesn't affect your typeset document in any way. Comments are useful for providing extra information about what's in your input file. For example:

```
% ========= Start of Section 'Hedgehog' =========
```

A comment starts with a percent sign (`%`) and extends to the end of the input line. TEX ignores not just the comment but the end of the line as well, so comments have another very important use: connecting two lines so that the end of line between them is invisible to TEX and doesn't generate an output space or an end of line. For instance, if you type:

```
A fool with a spread%
sheet is still a fool.
```

you'll get:

A fool with a spreadsheet is still a fool.

Punctuation

TEX normally adds some extra space after what it thinks is a punctuation mark at the end of a sentence, namely, '`.`', '`?`', or '`!`' followed by an input space. TEX doesn't add the extra space if the punctuation mark follows a capital letter, though, because it assumes the capital letter to be an initial in someone's name. You can force the extra space where it wouldn't otherwise occur by typing something like:

```
A computer from IBM\null?
```

The `\null` doesn't produce any output, but it does prevent TEX from associating the capital 'M' with the question mark. On the other hand, you can cancel the extra space where it doesn't belong by typing a control space after the punctuation mark, e.g.:

```
Proc.\␣Royal Acad.\␣of Twits
```

so that you'll get:

Proc. Royal Acad. of Twits

rather than:

Proc. Royal Acad. of Twits

Some people prefer not to leave more space after punctuation at the end of a sentence. You can get this effect with the `\frenchspacing` command (p. '`\frenchspacing`'). `\frenchspacing` is often recommended for bibliographies.

For single quotation marks, you should use the left and right single quotes (` and ') on your keyboard. For left and right double quotation marks, use two left single quotes or two right single quotes (`` or '') rather than the double quote (") on your keyboard. The keyboard double quote will in fact give you a right double quotation mark in many fonts, but the two right single quotes are the preferred TEX style. For example:

```
There is no `q' in this sentence.
``Talk, child,'' said the Unicorn.
She said, ``\thinspace`Enough!', he said.''
```

These three lines yield:

> There is no 'q' in this sentence.
> "Talk, child," said the Unicorn.
> She said, "'Enough!', he said."

The `\thinspace` in the third input line prevents the single quotation mark from coming too close to the double quotation marks. Without it, you'd just see three nearly equally spaced quotation marks in a row.

TEX has three kinds of dashes:

- Short ones (hyphens) like this (-). You get them by typing '`-`'.
- Medium ones (en-dashes) like this (–). You get them by typing '`--`'.
- Long ones (em-dashes) like this (—). You get them by typing '`---`'.

Typically you'd use hyphens to indicate compound words like "will-o'-the-wisp", en-dashes to indicate page ranges such as "pages 81–87", and em-dashes to indicate a break in continuity—like this.

Special characters

Certain characters have special meaning to TEX, so you shouldn't use them in ordinary text. They are:

```
$  #  &  %  _  ^  ~  {  }  \
```

In order to produce them in your typeset document, you need to use circumlocutions. For the first five, you should instead type:

```
\$  \#  \&  \%  \_
```

For the others, you need something more elaborate:

```
\^{␣}   \~{␣}   $\{$   $\}$   $\backslash$
```

Groups

A *group* consists of material enclosed in matching left and right braces (`{` and `}`). By placing a command within a group, you can limit its effects to the material within the group. For instance, the `\bf` command tells TEX to set something in **boldface** type. If you were to put `\bf` into your input and do nothing else to counteract it, everything in your document following the `\bf` would be set in boldface. By enclosing `\bf` in a group, you limit its effect to the group. For example, if you type:

```
We have {\bf a few boldface words} in this sentence.
```

you'll get:

We have **a few boldface words** in this sentence.

You can also use a group to limit the effect of an assignment to one of TEX's parameters. These parameters contain values that affect how TEX typesets your document. For example, the value of the `\parindent` parameter specifies the indentation at the beginning of a paragraph. The assignment `\parindent = 15pt` sets the indentation to 15 printer's points. By placing this assignment at the beginning of a group containing a few paragraphs, you can change the indentation of just those paragraphs. If you don't enclose the assignment in a group, the changed indentation will apply to the rest of the document (or up to the next assignment to `\parindent`, if there's a later one).

Not all pairs of braces indicate a group. In particular, the braces associated with an argument for which the braces are *not* required don't

indicate a group—they just serve to delimit the argument. Of those commands that do require braces for their arguments, some treat the braces as defining a group and the others interpret the argument in some special way that depends on the command.[1]

Math formulas

A math formula can appear in text (*text math*) or set off on a line by itself with extra vertical space around it (*display math*). You enclose a text formula in single dollar signs (`$`) and a displayed formula in double dollar signs (`$$`). For example:

```
If $a<b$, then the relation $$e^a < e^b$$ holds.
```

This input produces:

If $a < b$, then the relation

$$e^a < e^b$$

holds.

TEX does its own spacing inside math, ignoring any spaces in the input.

Section 'math' describes the commands that are useful in math formulas.

How TEX works

In order to use TEX effectively, it helps to have some idea of how TEX goes about its activity of transmuting input into output. You can imagine TEX as a kind of organism with "eyes", "mouth", "gullet", "stomach", and "intestines". Each part of the organism transforms its input in some way and passes the transformed input to the next stage.

The eyes transform an input file into a sequence of characters. The mouth transforms the sequence of characters into a sequence of *tokens*, where each token is either a single character or a control sequence. The gullet expands the tokens into a sequence of *primitive commands*, which are also tokens. The stomach carries out the operations specified by the primitive commands, producing a sequence of pages. Finally, the intestines transform each page into the form required for the `.dvi` file and

[1] More precisely, for primitive commands either the braces define a group or they enclose tokens that aren't processed in TEX's stomach. For `\halign` and `\valign` the group has a trivial effect because everything within the braces either doesn't reach the stomach (because it's in the template) or is enclosed in a further inner group.

send it there. These actions are described in more detail in Section 'concepts' under "anatomy of TEX" (p. '`\anatomy`').

The real typesetting goes on in the stomach. The commands instruct TEX to typeset such-and-such a character in such-and-such a font, to insert an interword space, to end a paragraph, and so on. Starting with individual typeset characters and other simple typographic elements, TEX builds up a page as a nest of boxes within boxes within boxes (see "box", p. '`box`'). Each typeset character occupies a box, and so does an entire page. A box can contain not just smaller boxes but also *glue* (p. '`glue`') and a few other things. The glue produces space between the smaller boxes. An important property of glue is that it can stretch and shrink; thus TEX can make a box larger or smaller by stretching or shrinking the glue within it.

Roughly speaking, a line is a box containing a sequence of character boxes, and a page is a box containing a sequence of line boxes. There's glue between the words of a line and between the lines of a page. TEX stretches or shrinks the glue on each line so as to make the right margin of the page come out even and the glue on each page so as to make the bottom margins of different pages be equal. Other kinds of typographical elements can also appear in a line or in a page, but we won't go into them here.

As part of the process of assembling pages, TEX needs to break paragraphs into lines and lines into pages. The stomach first sees a paragraph as one long line, in effect. It inserts *line breaks* in order to transform the paragraph into a sequence of lines of the right length, performing a rather elaborate analysis in order to choose the set of breaks that makes the paragraph look best (see "line break", p. '`line+break`'). The stomach carries out a similar but simpler process in order to transform a sequence of lines into a page. Essentially the stomach accumulates lines until no more lines can fit on the page. It then chooses a single place to break the page, putting the lines before the break on the current page and saving the lines after the break for the next page (see "page break", p. '`page+break`').

When TEX is assembling an entity from a list of items (boxes, glue, etc.), it is in one of six *modes* (p. '`mode`'). The kind of entity it is assembling defines the mode that it is in. There are two ordinary modes: ordinary horizontal mode for assembling paragraphs (before they are broken into lines) and ordinary vertical mode for assembling pages. There are two restricted modes: restricted horizontal mode for appending items horizontally to form a horizontal box and internal vertical mode for appending items vertically to form a vertical box (other than a page). Finally, there are two math modes: text math mode for assembling math formulas within a paragraph and display math mode for assembling math

formulas that are displayed on lines by themselves (see "Math formulas", p. ‘mathform’).

New TEX versus old TEX

In 1989 Knuth made a major revision to TEX in order to adapt it to the character sets needed to support typesetting for languages other than English. The revision included a few minor extra features that could be added without disturbing anything else. This book describes "new TEX". If you're still using an older version of TEX (version 2.991 or earlier), you'll want to know what features of new TEX you can't use. The following features aren't available in the older versions:

- `\badness` (p. ‘\badness’)
- `\emergencystretch` (p. ‘\emergencystretch’)
- `\errorcontextlines` (p. ‘\errorcontextlines’)
- `\holdinginserts` (p. ‘\holdinginserts’)
- `\language`, `\setlanguage`, and `\newlanguage` (pp. ‘\language’, ‘\@newlanguage’)
- `\lefthyphenmin` and `\righthyphenmin` (p. ‘\lefthyphenmin’)
- `\noboundary` (p. ‘\noboundary’)
- `\topglue` (p. ‘\topglue’)
- The `^^`*xy* notation for hexadecimal digits (p. ‘hexchars’)

We recommend that you obtain new TEX if you can.

Resources

A number of resources are available to help you in using TEX. *The TEXbook* is the definitive source of information on TEX:

> Knuth, Donald E., *The TEXbook*. Reading, Mass.: Addison-Wesley, 1984.

Be sure to get the seventeenth printing (January 1990) or later; the earlier printings don't cover the features of new TEX.

LATEX is a very popular collection of commands designed to simplify the use of TEX. It is described in:

> Lamport, Leslie, *The LATEX Document Preparation System*. Reading, Mass.: Addison-Wesley, 1986.

$\mathcal{AMS}$-TEX is the collection of commands adopted by the American Mathematical Society as a standard for submitting mathematical manuscripts electronically. It is described in:

Spivak, Michael D., *The Joy of TEX*. Providence, R.I.: American Mathematical Society, 1986.

You can join the TEX Users Group (TUG), which publishes a newsletter called *TUGBoat*. TUG is an excellent source not only for information about TEX but also for collections of macros, including $\mathcal{AMS}$-TEX. Its address is:

TEX Users Group
c/o American Mathematical Society
P.O. Box 9506
Providence, RI 02940
U.S.A.

Finally, you can obtain copies of the `eplain.tex` macros described in Section 'eplain' as well as the macros used in typesetting this book. They are available through the Internet network by anonymous `ftp` from the following hosts:

```
labrea.stanford.edu [36.8.0.47]
ics.uci.edu [128.195.1.1]
june.cs.washington.edu [128.95.1.4]
```

The electronic version includes additional macros that format input for the BIBTEX computer program, written by Oren Patashnik at Stanford University, and print the output from that program. If you find bugs in the macros, or think of improvements, you can send electronic mail to Karl at `karl@cs.umb.edu`.

The macros are also available for US $10.00 on $5\frac{1}{4}''$ or $3\frac{1}{2}''$ PC-format diskettes from:

Paul Abrahams
214 River Road
Deerfield, MA 01342
Email: `Abrahams%Wayne-MTS@um.cc.umich.edu`

These addresses are correct as of June 1990; please be aware that they may change after that, particularly the electronic addresses.